Software Architecture and Design Specification

Project: Resource Usage Monitor (ReMo)

Version: 1.0

Authors: Shreyas GS, Unnathi Baskar, Uday Gopan, Vishnu Prasath R S

Date: 26-09-2025

Status: Draft

Revision History

Version	Date	Author	Summary
1.0	26-09-2025	Team	Initial SAD document for ReMo system

Approvals

Role	Name	Signature/Date
QA Lead	Unnathi Baskar	
Dev Lead	Uday Gopan	
Product Owner	Prof. Sheela Devi	

1. Introduction

1.1 Purpose

This document specifies the architecture and design of the Resource Usage Monitor (ReMo) system. The ReMo system is designed to provide a dashboard that displays server resource usage (CPU and memory) over time using mock metric-generation stubs, visualize trends, trigger alerts when thresholds are exceeded, and help users understand how monitoring and alerting systems work.

1.2 Scope

The ReMo system covers the following key functionalities:

- Display mock CPU and memory metrics over time using metric generation stubs
- Provide a customizable dashboard with charts and graphs for visualization
- Generate alerts/notifications when user-defined thresholds are exceeded
- Support user authentication and node management
- Persistent storage for historical data and logs
- Data export functionality in multiple formats (CSV, JSON, XML)
- Configuration options for thresholds, refresh intervals, and dashboard layout

1.3 Audience

This document is intended for:

- Developers/Students implementing the system
- QA engineers and testers validating the system
- System administrators managing deployments
- Project evaluators and instructors reviewing the system
- Maintenance teams supporting the system

1.4 Definitions

- **ReMo**: Resource Usage Monitor system
- **CPU**: Central Processing Unit usage metrics
- **Memory Usage**: RAM utilization metrics
- **Metric Stub**: Mock data generation component simulating server metrics
- **Charting Library**: Visualization component for rendering time-series graphs
- **Dashboard**: Main user interface displaying metrics and alerts
- **Threshold**: User-configurable limits that trigger alerts when exceeded
- **Node**: Individual server or system being monitored
- **GUI**: Graphical User Interface

- **API**: Application Programming Interface

2. Document Overview

2.1 How to use this document

This document provides comprehensive architectural deliverables including:

- System architecture patterns and rationale
- Component descriptions and UML diagrams
- API design specifications
- Security architecture and threat modeling
- Technology stack decisions
- Traceability to requirements from SRS

2.2 Related Documents

- SRS v1.0: Resource Usage Monitor (Software Requirements Specification)
- STP v1.0: Resource Usage Monitor (Software Test Plan)
- RTM: Requirements Traceability Matrix

3. Architecture

3.1 Goals & Constraints

Goals:

- Provide demonstration of monitoring systems
- Maintain lightweight and responsive performance (90% actions <2 seconds)
- Ensure reliable operation (>99.9% uptime for 7+ days)
- Support scalability for >10 servers/metric types
- Deliver intuitive user experience with clear alert visualization
- Maintain modular and maintainable codebase

Constraints:

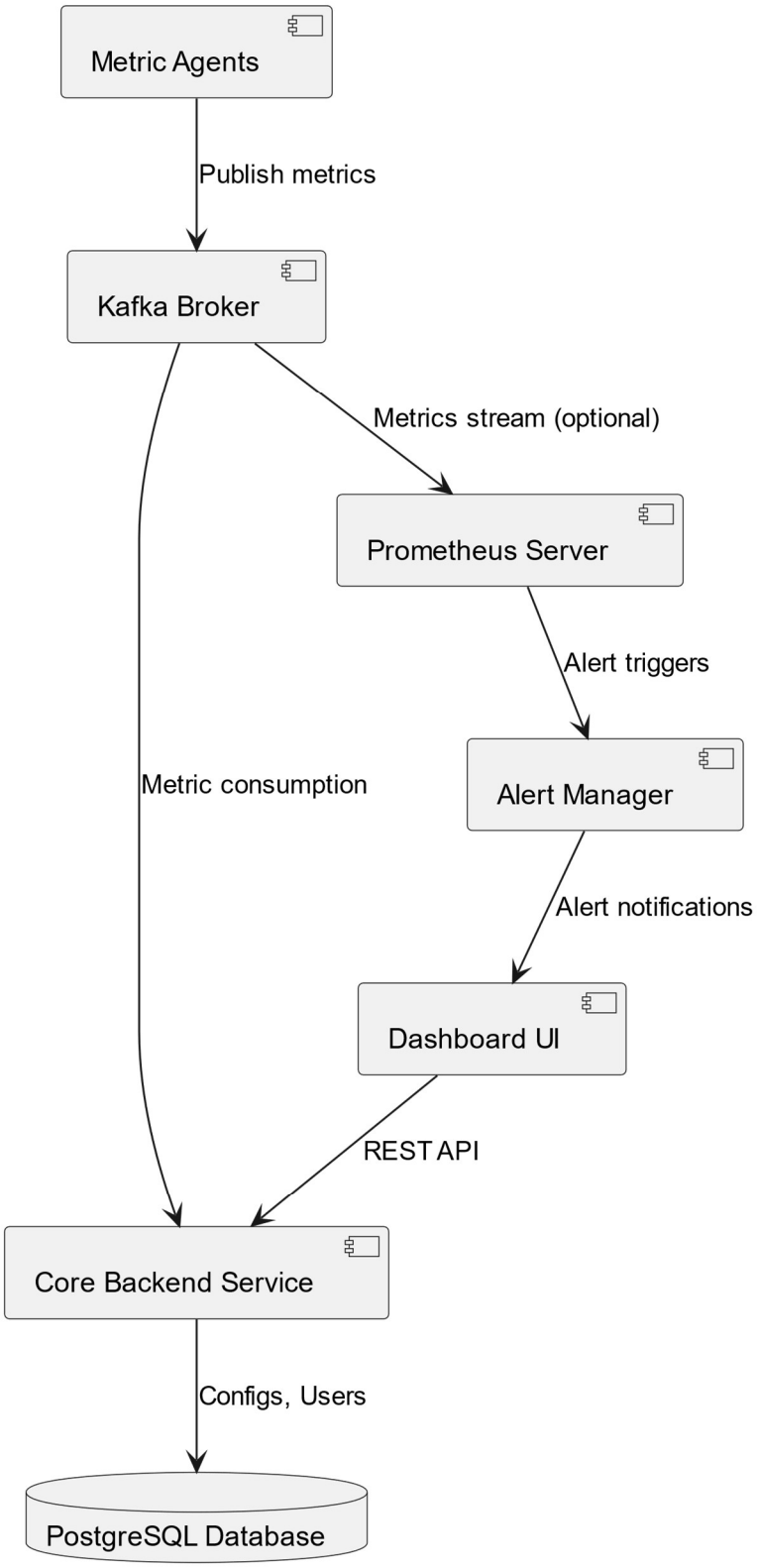
- System uses simulated data only (no actual hardware monitoring)

- English language support only in v1.0
- Minimum hardware requirements: 2GB RAM, dual-core CPU
- Linux Ubuntu 20.04+ operating system requirement
- Must comply with basic security practices (encrypted storage, secure communications)

3.2 Stakeholders & Concerns

General Users: Security, availability, intuitive dashboard visualization
Developers/Testers: Modularity, maintainability, clear interfaces, extensibility
System Administrators: Reliability, performance, easy deployment
Evaluators/Instructors: Educational value, correctness, comprehensive feature coverage
Security: Authentication, authorization, audit logging, data protection

3.3 Component (UML) Diagram



3.4 Component Descriptions

- **Dashboard UI:** Provides the visual interface for users to view real-time and historical server CPU and memory usage, alerts, and configurations.

- **Metric Agents:** Collect CPU and memory metrics from monitored servers and publish these as messages to Kafka topics.

- **Kafka Broker:** Serves as a durable, distributed message queue receiving metrics from agents, decoupling metric generation from processing and enabling asynchronous scalable ingestion.

- **Prometheus Server:** Optionally scrapes metrics from agents or consumes from Kafka via connectors, stores the time-series data, and evaluates alert rules.

- **Alert Manager:** Handles alert routing and notifications based on alert conditions from Prometheus.

- **Core Backend Service:** Consumes metrics from Kafka, manages user authentication, configuration data, alert rule management, and serves API endpoints for the dashboard.

- **Database:** Stores user data, alert configurations, server metadata, and audit logs in a relational database (e.g., PostgreSQL).

3.5 Chosen Architecture Pattern and Rationale

Hybrid Modular Monolith Architecture with event-driven ingestion via Kafka

Chosen for balanced modularity, scalability, and ease of deployment. Kafka decouples agents and backend for reliable, scalable metric ingestion compared to synchronous HTTP. Prometheus remains the specialized time-series data store and alerting engine. This pattern avoids microservices complexity while supporting growth and fault tolerance.

3.6 Technology Stack & Data Stores

- **Backend:** Python (FastAPI) with Kafka consumer clients; Prometheus for time-series monitoring and alerting.
- **Frontend:** Svelte
- **Storage:** Prometheus time-series database; PostgreSQL for relational data management.
- **Messaging:** Apache Kafka as central message bus for metrics ingestion.
- **Security:** TLS encryption for all network communication; OAuth 2.0 / OpenID Connect for authentication and authorization.

3.7 Risks & Mitigations

Risk	Impact	Mitigation Strategy
Data loss or delay due to network failure	High	Kafka's durable buffering and retry mechanisms.
Unauthorized access	High	Strong authentication (OAuth 2.0), RBAC authorization.
Alert flooding	Medium	Alert throttling and aggregation in Alertmanager.
Performance Bottlenecks	Medium	Scale Kafka clusters, backend consumers, optimize Prometheus data retention and query.

3.8 Traceability to Requirements

Authentication Requirements:

- FR-1.1 (User Authentication - OAuth2 / Username/Password) → Core Backend Service (Authentication Module)
- FR-1.2 (API Access Control) → API Layer in Core Backend Service with OAuth 2.0 and RBAC

- FR-1.3 (Account Lockout / Security Policies) → Core Backend Service (Authentication Module)

- FR-1.4 (Agent Authentication Tokens) → Metric Agents + Kafka with secure token-based authentication

Dashboard Requirements:

- FR-2.1 (Real-time and Customizable Dashboard Layout) → Dashboard UI (Svelte) + Core Backend API

- FR-2.2 (User Settings and Alert Configuration Management) → Core Backend Service (Configuration Module)

Data Management Requirements:

- FR-3.1 (Mock Metric Generation for Testing) → Metric Agents with Mock Metric Mode

- FR-3.2 (Selection of Monitored Servers / Nodes) → Core Backend Service + Dashboard UI

- FR-3.3 (Metric Data Publishing) → Metric Agents publishing to Kafka Topics

- FR-3.4 (Metric Data Consumption and Preprocessing) → Core Backend Service Kafka Consumer + Prometheus Scraper

- FR-3.5 (Alert Rule Evaluation and Data Processing) → Prometheus Server + Alert Manager

Alerting Requirements:

- FR-4.1 (Customizable Threshold and Alert Settings) → Core Backend Service (Alert Configuration Module) + Prometheus Alert Manager

- FR-4.2 (Alert Notifications and Display) → Alert Manager + Dashboard UI

Storage & Visualization Requirements:

- FR-5.1 (Durable Time-Series Storage of Metrics) → Prometheus Time-Series Database
- FR-5.2 (Current Metrics Visualization) → Dashboard UI + Core Backend API querying Prometheus
- FR-5.3 (Historical Metrics Visualization) → Dashboard UI + Prometheus
- FR-6.1 (Export of Metrics and Reports) → Core Backend Export Service

3.9 Security Architecture

- **Spoofing:** Secure agent authentication via token or mutual TLS.
- **Tampering:** Integrity checks in agent metrics; secure backend API endpoints.
- **Information Disclosure:** Enforce TLS encryption for all communications between agents, Prometheus, backend, and UI.
- **Denial of Service (DoS):** Rate limiting on API endpoints and alert notification throttling.
- **Elevation of Privilege:** Role-Based Access Control (RBAC) enforced at API and UI layers; principle of least privilege applied.

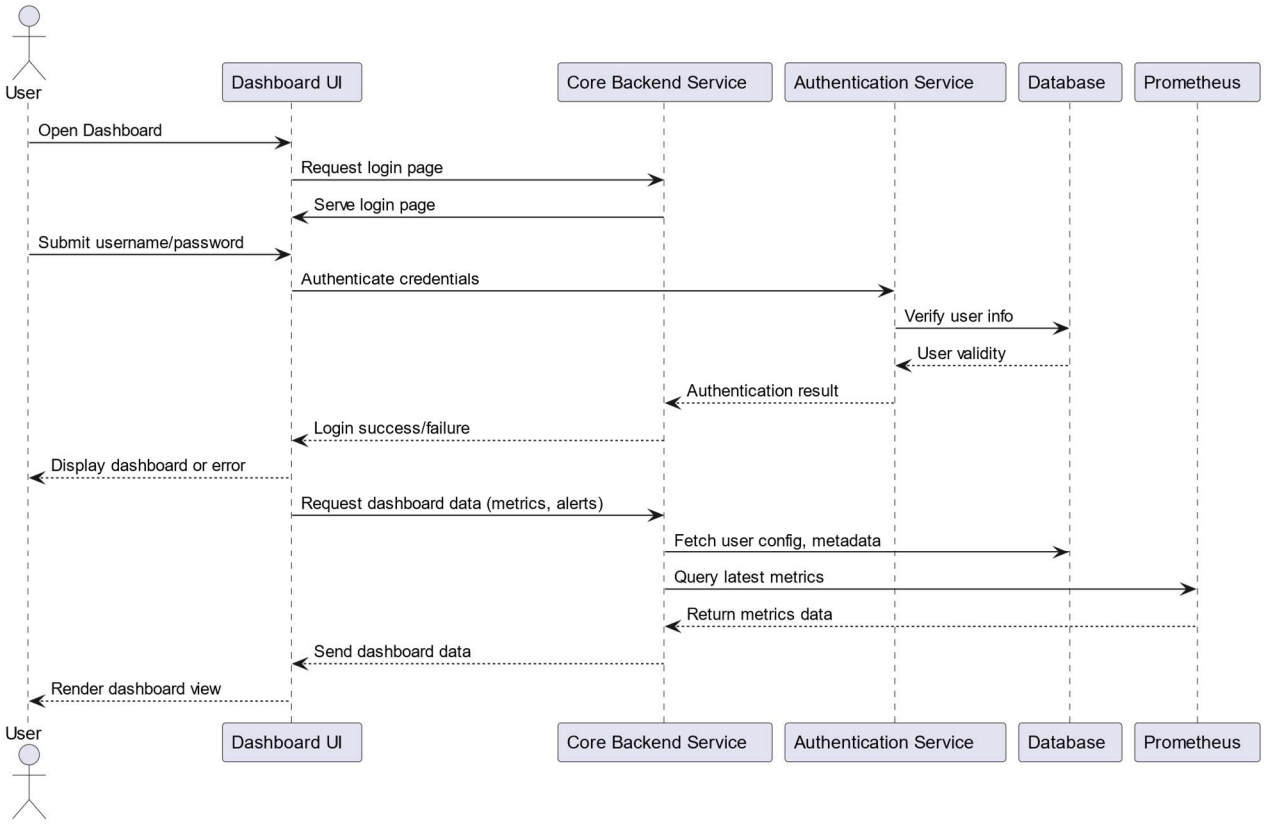
4. Design

4.1 Design Overview

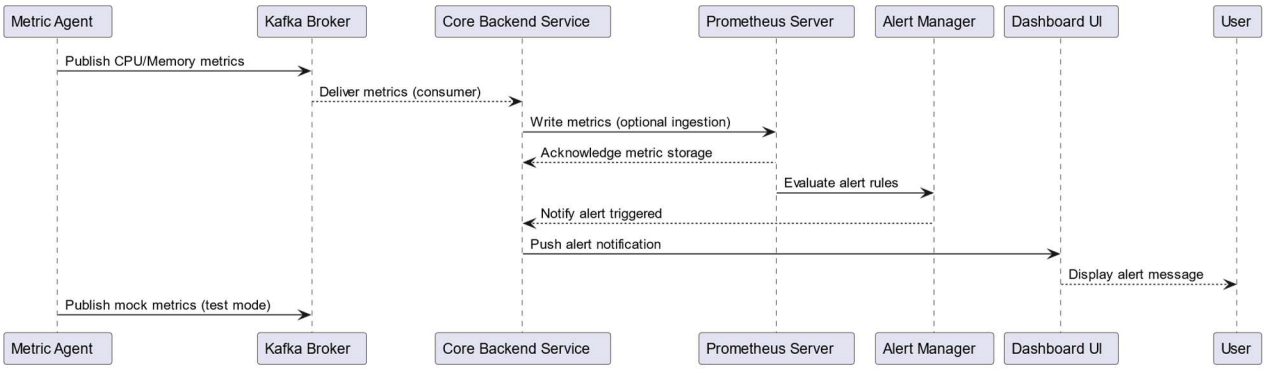
- **User-Centric Interface**: Intuitive dashboard with customizable layouts
- **Data Flow Architecture**: Clear path from metric generation through processing to visualization
- **Real-Time Processing**: Live metric updates with configurable refresh intervals
- **Historical Analysis**: Time-based data storage and retrieval for trend analysis
- **Alert Management**: Proactive notification system with threshold-based triggers
- **Security-First Design**: Comprehensive authentication, authorization, and audit capabilities

4.2 UML Sequence Diagrams

Sequence Diagram 1: User Authentication and Dashboard Access



Sequence Diagram 2: Metric Collection and Alert Processing



4.3 API Design

Authentication API

Login:
Endpoint: /api/auth/login
Method: POST
Request:
```
{
  "username": "string",
  "password": "string"
}
```
Response:
```
{
  "status": "success",
  "token": "string",
  "expires_in": 3600
}
```

Errors:

401 Unauthorized – Invalid credentials

423 Locked – Account locked

429 Too Many Requests – Rate limit exceeded

Logout:
Endpoint: /api/auth/logout
Method: POST
Headers: Authorization: Bearer {token}
Response:
```
{
  "status": "success",
  "message": "Logged out successfully"
}
```

Metrics API

Get Current Metrics:
Endpoint: /api/metrics/current
Method: GET

Headers: Authorization: Bearer {token}
Query Parameters:

node_id (optional): Filter by specific node

timeframe (optional): Time window in minutes (default last 5 mins)
Response:

```
{
  "timestamp": "2025-09-26T17:30:00Z",
  "metrics": {
    "cpu_usage": 75.5,
    "memory_usage": 62.3,
    "node_id": "node-001"
  }
}
```

Get Historical Metrics:
Endpoint: /api/metrics/historical
Method: GET
Headers: Authorization: Bearer {token}
Query Parameters:

start_date: ISO 8601 date string (required)

end_date: ISO 8601 date string (required)

node_id (optional): Filter by node
Response:

```
{
  "data": [
    {
      "timestamp": "2025-09-26T17:00:00Z",
      "cpu_usage": 68.2,
      "memory_usage": 58.7,
      "node_id": "node-001"
    }
  ],
  "total_records": 1440
}
```

Threshold Management API

Create Threshold:
Endpoint: /api/thresholds
Method: POST

Headers: Authorization: Bearer {token}
Request:

```
{
  "metric_type": "cpu_usage",
  "threshold_value": 80.0,
  "comparison": "greater_than",
  "node_id": "node-001"
}
```

Response:

```
{
  "status": "success",
  "threshold_id": "th-001",
  "message": "Threshold created successfully"
}
```

List Thresholds:
Endpoint: /api/thresholds
Method: GET
Headers: Authorization: Bearer {token}
Response:

```
{
  "thresholds": [
    {
      "threshold_id": "th-001",
      "metric_type": "cpu_usage",
      "threshold_value": 80.0,
      "comparison": "greater_than",
      "node_id": "node-001"
    }
  ]
}
```

Export API

Export Metrics Data:
Endpoint: /api/export/data
Method: POST
Headers: Authorization: Bearer {token}
Request:

```
{
  "start_date": "2025-09-25T00:00:00Z",
  "end_date": "2025-09-26T23:59:59Z",
  "format": "csv",
  "node_ids": ["node-001", "node-002"]
}
```

Response: File download or

```
{
```

```
  "download_url": "/api/downloads/export-12345.csv",
  "expires_at": "2025-09-27T17:30:00Z"
}
```


4.4 Error Handling, Logging & Monitoring

Error Handling Strategy:

- Standardized HTTP status codes for API responses
- Detailed error messages without exposing sensitive system information
- Graceful degradation for non-critical failures
- User-friendly error messages in the frontend
- Automatic retry mechanisms for transient failures

Logging Framework:

- Structured logging with JSON format for easy parsing
- Log levels: DEBUG, INFO, WARN, ERROR, FATAL
- No sensitive information (passwords, tokens) in logs
- Correlation IDs for tracing requests across services
- Log rotation and retention policies

Security Logging:

- All authentication attempts (successful and failed)
- Authorization failures and access violations
- Configuration changes and administrative actions
- Alert triggers and threshold modifications
- System startup and shutdown events

Monitoring Metrics:

- **Performance**: API response times, database query performance
- **System Health**: Memory usage, CPU utilization, disk space
- **Business Metrics**: Alert frequency, threshold violations, user activity
- **Security Metrics**: Failed login attempts, suspicious activities

- **Availability**: Service uptime, error rates, system availability

Alerting Thresholds:

- API response time > 2 seconds for 90th percentile
- Error rate > 5% over 5-minute window
- Failed authentication attempts > 10 per minute from single IP
- System memory usage > 90%
- Database connection failures

4.5 UX Design

Dashboard Design Principles:

- **Clarity**: Clean, uncluttered interface with clear visual hierarchy
- **Responsiveness**: Adaptive layout supporting different screen sizes
- **Accessibility**: WCAG 2.1 AA compliance for users with disabilities
- **Performance**: Fast loading with progressive data display
- **Customization**: Drag-and-drop widgets with resizable components

User Interface Components:

- **Navigation Bar**: Quick access to main features and user settings
- **Metric Widgets**: Configurable charts displaying CPU and memory usage
- **Alert Panel**: Prominent display of active alerts with severity indicators
- **Configuration Panel**: Intuitive threshold and settings management
- **Historical View**: Time range selector with zoom and pan capabilities
- **Export Interface**: Simple data export with format selection

Alert Visualization:

- **Color Coding**: Green (normal), Yellow (warning), Red (critical)
- **Visual Indicators**: Icons, badges, and status bars for quick recognition
- **Sound Alerts**: Optional audio notifications for critical alerts
- **Alert History**: Chronological list with acknowledgment tracking

- **Contextual Information**: Detailed alert descriptions with recommended actions

Accessibility Features:

- High contrast mode support
- Keyboard navigation compatibility
- Screen reader compatibility with ARIA labels
- Scalable fonts and interface elements
- Alternative text for visual elements

4.6 Open Issues & Next Steps

Current Limitations:

- English language support only (internationalization needed)
- Simulated data only (real hardware monitoring integration required)
- Single server deployment (multi-server architecture needed)
- Basic authentication (SSO and MFA integration planned)

Future Enhancements:

- **Real Hardware Integration**: Connect to actual system metrics using OS APIs
- **Multi-Server Support**: Distributed monitoring across multiple servers
- **Advanced Analytics**: Predictive alerting and anomaly detection
- **Mobile Application**: Native mobile app for on-the-go monitoring
- **Integration APIs**: Third-party monitoring tool integrations
- **Machine Learning**: Intelligent threshold recommendations
- **Custom Dashboards**: User-defined dashboard templates and sharing

Technical Debt:

- Database optimization for large-scale historical data
- Caching layer implementation for improved performance
- Containerization for easier deployment and scaling
- CI/CD pipeline establishment for automated testing and deployment

Security Enhancements:

- Multi-factor authentication implementation
- OAuth 2.0 integration for enterprise SSO
- Advanced threat detection and prevention
- Compliance certification (SOC 2, ISO 27001)

5. Appendices

5.1 Glossary

API: Application Programming Interface - standardized interface for software communication
Audit Trail: Chronological record of system activities for security and compliance
CPU: Central Processing Unit - primary computation component of a computer
Dashboard: Visual interface displaying key metrics and system information
JSON: JavaScript Object Notation - lightweight data interchange format
Metric: Quantitative measurement of system performance or resource usage
Node: Individual server or system component being monitored
REST: Representational State Transfer - architectural style for web services
SLA: Service Level Agreement - formal commitment to system performance levels
Threshold: Predefined limit that triggers alerts when exceeded
TLS: Transport Layer Security - cryptographic protocol for secure communication
UML: Unified Modeling Language - standardized modeling notation for software design

5.2 References

- **IEEE 42010**: Systems and software engineering - Architecture description
- **OWASP Top 10**: Web application security risks and mitigation strategies
- **NIST SP 800-160**: Systems Security Engineering framework
- **WCAG 2.1**: Web Content Accessibility Guidelines
- **REST API Design Guidelines**: Industry best practices for RESTful services
- **Python Security Best Practices**: Secure coding standards for Python applications

5.3 Tools

Design & Documentation:

- Draw.io: Visual architecture and system diagrams
- Swagger/OpenAPI: API documentation and testing

Development & Testing:

- Selenium: Automated web application testing
- Postman: API development and testing platform
- curl: Command-line HTTP client for API validation
- pytest: Python testing framework

Monitoring & Performance:

- Custom performance testing scripts
- Logging utilities for audit verification
- System monitoring tools for resource tracking